Cache-Lab 实验报告

21307130028 沈钰

一、实现思路：

PartA：模拟高速缓存

1.定义高速缓存行结构体 cache_line

定义有效位 v，标记位 tag，B 不用处理因此不定义。由于缓存替换策略是 LRU，因此定义每个缓存行的时间戳，以此来决定驱逐哪一行。

```
typedef struct{          //定义高速缓存行
    int valid_bits;      //有效位
    unsigned  tag;       //标记位
    int stamp;           //时间戳，用于LRU
}cache_line;
```

2.初始化 cache

将 cache 定义为 cache_line 类型的二重数组，并用 malloc()函数为每一个 cache_line 分配内存空间，并将有效位和时间戳初始化为 0，tag 初始化为无效位。

```
cache_line** cache = NULL;      //cache[S][E]，二级指针

void init_cacheline(){
    cache = (cache_line**)malloc(sizeof(cache_line*)*S);            //为cache组分配空间
    for(int i=0;i<S;i++)
        *(cache+i) = (cache_line*)malloc(sizeof(cache_line)*E);      //每一组为cache行分配空间
    for(int i=0;i<S;i++){
        for(int j=0;j<E;j++){
            cache[i][j].valid_bits = 0;      //有效位初始化为0
            cache[i][j].tag = 0xffffffff;    //tag初始化为无效位
            cache[i][j].stamp = 0;           //时间戳为0
        }
    }
}
```

3.解析指令

使用讲义中建议的 getopt()来解析 linux 指令，获取 s,E,b 和 trace 文件地址。

```
int main(int argc,char *argv[])                 //解析输入的指令
{
    int opt;
    while((opt = getopt(argc,argv,"s:E:b:t:")) !=-1){          //parse command line arguments
        switch(opt){
        case 's':
            s=atoi(optarg);                         //s,E,b分别表示组索引位数，行数，块位数
            break;
        case 'E':
            E=atoi(optarg);
            break;
        case 'b':
            b=atoi(optarg);
            break;
        case 't':                                   //要重播的valgrind跟踪的名称
            filepath = optarg;
            break;
        }
    }
```

4.读取 trace 文件中的操作

使用讲义中建议的 fscanf()来解析操作出 L,M 和 S 和地址 address

```
while(fscanf(file," %c %x,%d",&operation,&address,&size)>0){   //读取文件中的操作，无需考虑I
    switch(operation){
        case 'L':
            update_cache(address);
            break;
        case 'M':
            update_cache(address);
        case 'S':
            update_cache(address);
            break;
    }
    update_time();                  //时间戳更新
}
```

5.更新 cache

每个操作都需要我们更新 cache。根据现有 cache 情况和 tag 对应结果，分为命中、冷不命中（cold miss）和冲突不命中（conflict miss）三种情况。

从 address 中解析出组索引 s 和标记位 t。在对应组中扫描所有行，比较 t 和缓存行 tag。

t 等于 tag 则为命中（hit）。累计 hit 数，并刷新时间戳。

t 不等于 tag 时：

①如果组中**存在有效位 v 为 0 的行，则为冷不命中**（cold miss），在更新该行标记位并将有效位置 1。累计 miss 数，并刷新时间戳。

②**若无有效位 v 为 0 的行，则为冲突不命中**（conflict miss）。由于替换策略时 LRU，我们**驱逐出时间戳 stamp 最大的一行**，并更新为地址的标记位。累计 miss 数和 eviction 数，并刷新时间戳。

以上操作结束后，更新所有有效位为 1 的 cache 行的时间戳。

6.总结

将以上部分组合起来，即可分析所有 trace 文件中的操作并且更新 cache。最后用 printSummary()打印 hit、miss、eviction 数。

PartB：转置函数 cache miss 优化

思路：示例的 trans()函数 miss 数高的原因是，虽然数组 A 是元素按行访问，在内存上连续，但转置到的数组 B 是按列访问的，访问模式的步长很大，会导致许多缓存未命中。因此，主要的优化思路是通过分块来增加空间局部性，提高缓存行的命中率。已知的条件是：cache 直接相联映射，一共 32 组（行），每行存 32 个字节，这意味着每行可以存 8 个数组 int 元素。

1.M = 32， N = 32

一个 cacheline 可以存 8 个内存连续的元素。则对于该矩阵的一行，需要 4 个 cacheline。这意味着整个 cache 可以刚好存矩阵中 8 行的所有元素。

因此选择 8*8 的分块进行转置，这样每次访问 A 时，都可以完全利用一个 cacheline 的 8 个元素再丢弃；每次访问 B 时，需要的元素都在一个 cacheline 中，不会频繁的替换即 cache 抖动。

对于矩阵对角线上那些元素来说，在 A 中的位置等于在 B 中的位置，它们在转置的过程中，访问 A 和访问 B 映射的是同一个 cacheline，这里就会发生冲突不命中，发生替换和二次载入。因此，**利用局部变量将一个 cacheline 中 8 个元素存入寄存器**，这样访问 A 的 cacheline 只要载入一次即可，避免再次载入。

因此，采用 8*8 分块和 8 个临时变量来协助转置。

2.M = 64， N=64

N = 64 意味矩阵一行需要 8 个 cacheline，整个 cache 只能存矩阵中 8 行的所有元素。这意味着如果对 B 的访问跨度超过 4 行，就会发生 cache 抖动。

因此，在之前 8*8 分块和 8 个临时变量的基础上，**将 8*8 分块分划为 4 个 4*4 的小块进行分步处理，这里运用了分块转置的思路，较为复杂：**

①读 A4 行（4*8），转置一个小块到（B 左上），转置保存一个小块（到 B 右上）

②B 右上小块移动到左下，读取 A 左下小块转置到 B 右上

③转置 A 右下小块到 B 右下

A

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

B

1	9	17	25	5	13	21	29
2	10	18	26	6	14	22	30
3	11	19	27	7	15	23	31
4	12	20	28	8	16	24	32
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	0	0	0	0
6	14	22	30	0	0	0	0
7	15	23	31	0	0	0	0
8	16	24	32	0	0	0	0

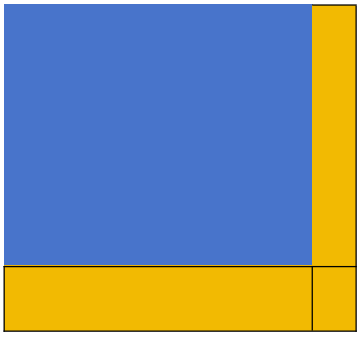
1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63
8	16	24	32	40	48	56	64

①中的操作将一整行的 8 个 int 都用到，提高 cacheline 使用的效率，避免了再次载入。

3.M = 61，N = 67

这样的矩阵规格是不太规律的，8*8 分块可能未必是最好的分块结果。但是 8*8 分块的优势在于可以利用局部变量减少对角线元素的重新载入 cache，且能更好利用读取 A 得到的 cacheline。

因此将矩阵分割成若干 8*8 块进行转置操作，然后再转置右下角、右方和下方剩余的部分。



二、实验结果

```
21307130028@5c7e868196b8:~/cachelab-handout$ ./driver.py
Part A: Testing cache simulator
Running ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     3 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace
     3 (4,2,4)       4       5       2       4       5       2  traces/yi.trace
     3 (2,1,4)       2       3       1       2       3       1  traces/dave.trace
     3 (2,1,3)     167      71      67     167      71      67  traces/trans.trace
     3 (2,2,3)     201      37      29     201      37      29  traces/trans.trace
     3 (2,4,3)     212      26      10     212      26      10  traces/trans.trace
     3 (5,1,5)     231       7       0     231       7       0  traces/trans.trace
     6 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trace
    27


Part B: Testing transpose function
Running ./test-trans -M 32 -N 32
Running ./test-trans -M 64 -N 64
Running ./test-trans -M 61 -N 67

Cache Lab summary:
                        Points   Max pts      Misses
Csim correctness          27.0        27
Trans perf 32x32           8.0         8         287
Trans perf 64x64           8.0         8        1179
Trans perf 61x67          10.0        10        1905
          Total points    53.0        53
```

partA：模拟器得到的结果均准确，得到 27 分满分。

partB：三种规格的结果分别为 287、1179、1905 次，均为满分。